Lecture 3: Interacting with the OS

OS Strucutre, System Call, Objects, File Descriptor, and Pipes

Xin Liu

Florida State University

xliu15@fsu.edu

COP 4610 Operating Systems

https://xinliulab.github.io/FSU-COP4610-Operating-Systems/

Recap 1: Program Pipeline

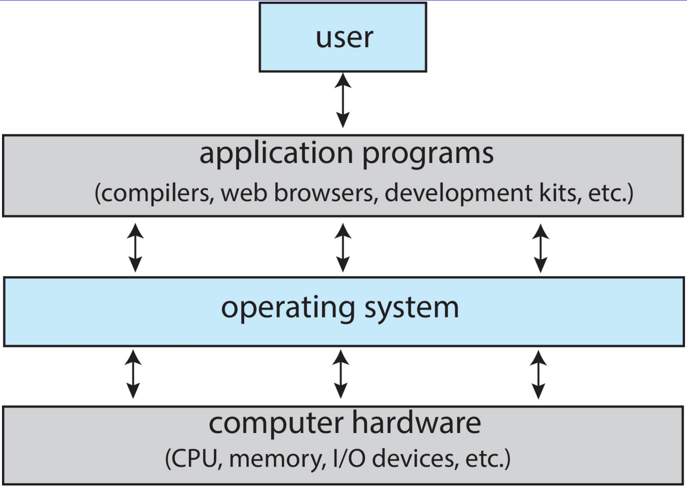
- You write a program, for example `HelloWorld.c`.
- The code goes through four steps: preprocessing, compilation, assembly, and linking.
  - The compiler turns C into assembly and applies optimization.
  - Optimization may reorder or change instructions, so the final machine code is not predictable from the source alone.
  - Think of two “compilers”: the compiler changes your program, and the CPU changes the execution order with reordering and speculation.
- The computer follows the optimized machine code.
- The computer is always right!
- Tip: **Never assume the computer runs your code in the order you wrote.**

Recap 2: Program Boundary

- A program is a state machine.
- Running code means state transitions.
- Your code can only change its own internal state.
  - Example: your `HelloWorld.c` prints text, then it exits because the OS already sets up `_start` and call `exit` for you.
  - If you write only `_start` and use no libraries, nothing will call `exit`. You must make the `exit` system call yourself.
- Anything outside the program state is done by the OS through a system call.

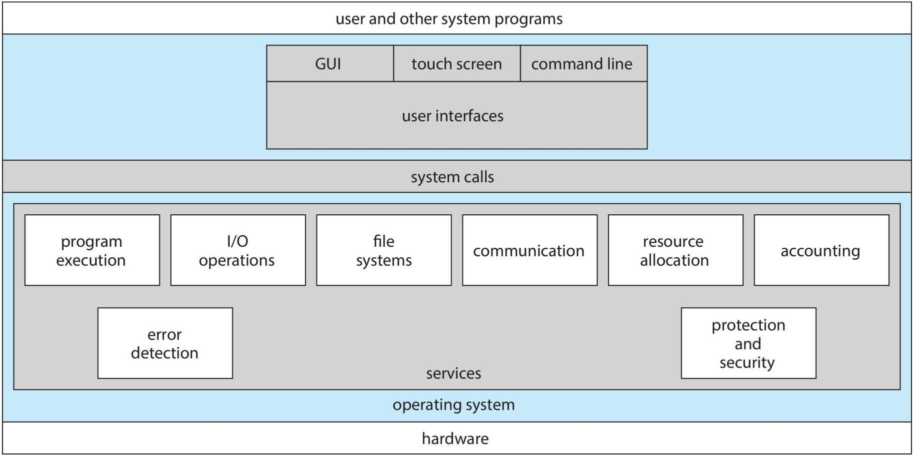
OS Structure

Abstract View of Components of Computer



- OS is a program that acts as an intermediary between a user of a computer and the computer hardware.
- OS hides the complexity and limitations of hardware from application programmers

A View of Operating System Services



Operating System Services

- **User interface**
  - Almost all operating systems provide a user interface (UI).
  - Forms: command-line (CLI), graphical (GUI), touch-screen, batch.
- **Program execution**
  - Load a program into memory, run it, then terminate.
  - Termination can be normal or abnormal (error).
- **I/O operations**
  - A running program may request I/O.
  - I/O may involve files or external devices.
- **File-system manipulation**
  - Read and write files and directories.
  - Create and delete files and directories.
  - Search, list file information, and manage permissions.

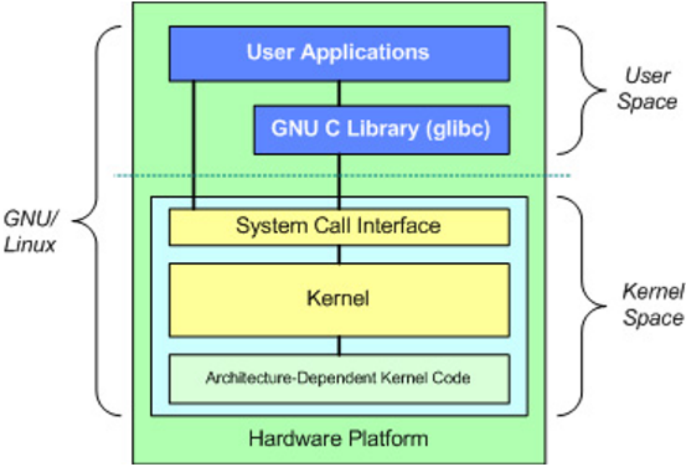
Operating System Services (Cont.)

- **Communications**
  - Processes exchange information on the same computer or across a network.
  - Methods include shared memory and message passing (packets moved by the OS).
- **Error detection**
  - Errors can occur in the CPU, memory, I/O devices, or user programs.
  - The OS takes appropriate actions to keep computing correct and consistent.
  - Debugging facilities help users and programmers work efficiently.
- **Resource allocation**
  - With multiple users or concurrent jobs, the OS allocates resources to each.
  - Resources include CPU time, main memory, file storage, and I/O devices.

Operating System Services (Cont.)

- **Logging**
  - Track which users use which resources and how much.
- **Protection and security**
  - Control access to information in multiuser or networked systems.
  - Ensure concurrent processes do not interfere with each other.
  - Protection means all access to system resources is controlled.
  - Security requires user authentication and defends external devices from invalid access.

OS Architecture: Kernel Space and User Space



Kernel Space V.s. User Space

Kernel space

- The OS kernel runs here with the highest privilege.
- **Function:** Manage CPU, memory, and device drivers.
- **Security:** Kernel bugs can crash the whole system.
- **Access:** Only kernel mode code can touch kernel memory.

User space

- Regular applications run here with lower privilege.
- **Function:** Run apps such as browsers and editors.
- **Security:** App bugs usually crash only that process.
- **Access:** User code cannot access kernel memory, it must use system calls.

OS Structure

- A general-purpose OS is a very large program.
- There are several common structures:
  - **Simple structure** (MS-DOS).
  - **Monolithic UNIX** (more complex but modular).
  - **Layered structure** (build abstractions in layers).
  - **Microkernel** (e.g., Mach).

Monolithic Structure: The Original UNIX Design

- UNIX was limited by the hardware of its time.
- The original UNIX used a simple structure with minimal organization.
- Two parts:
  - System programs.
  - The kernel.
- The kernel:
  - Sits between system calls and the hardware.
  - Manages the file system, CPU scheduling, memory, and other OS functions in one layer.

Traditional UNIX System Structure

Beyond simple, not fully layered.

(the users)

shells and commands
compilers and interpreters
system libraries

system-call interface to the kernel

signals terminal handling character I/O system terminal drivers	file system swapping block I/O system disk and tape drivers	CPU scheduling page replacement demand paging virtual memory

kernel interface to the hardware

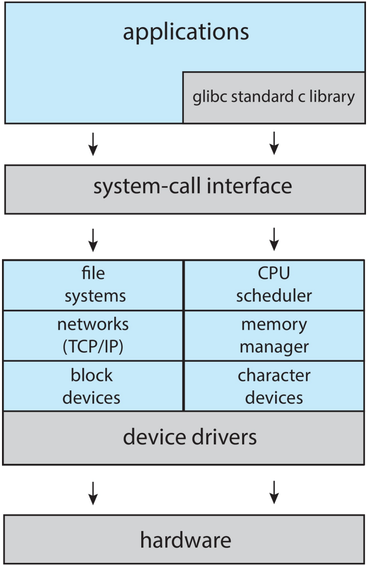
terminal controllers terminals	device controllers disks and tapes	memory controllers physical memory

kernel

Dennis Ritchie stands over Ken Thompson as he works on the PDP-11 in 1972. Courtesy Bell Labs

Linux System Structure

Monolithic plus modular design

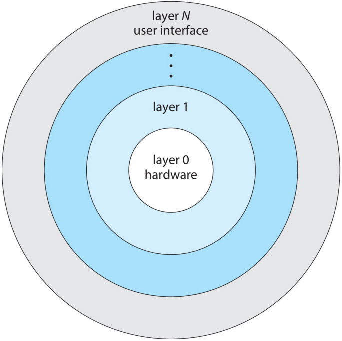


Linus Torvalds **Quote**
"As Sara and I used to say, just give Linus a spare closet with a good computer in it and feed him some dry pasta, and he'll be perfectly happy." (Just for Fun: The Story of an Accidental Revolutionary. Linus Torvalds and David Diamond. HarperBusiness, 2001 (paperback 2002).)

Curiosity Is All You Need.
Attention Is All You Need As Well.

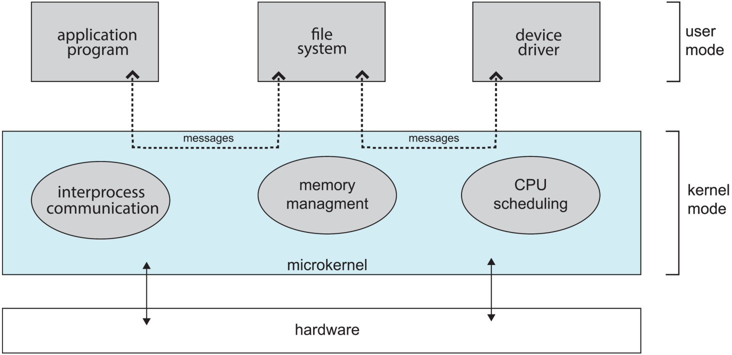
Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers.
- The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers.



Microkernels

- Moves as much from the kernel into user space.
- Communication takes place between user modules using message passing.
- Benefits: Easier to extend a microkernel, easier to port the operating system to new architectures, more reliable, and more secure (less code is running in kernel mode).
- Detriments: Performance overhead of user space to kernel space communication



Modules

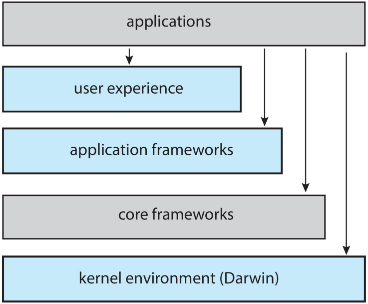
- Many modern operating systems implement loadable kernel modules (LKMs).
    - Uses object-oriented approach
    - Each core component is separate
    - Each talks to the others over known interfaces
    - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible
    - Linux, Solaris, etc.

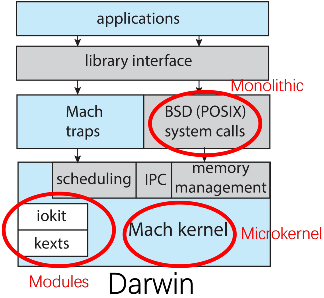
Hybrid Systems

- Most modern operating systems are not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic, plus microkernel for different subsystem personalities
- Apple Mac OS X hybrid, layered, Aqua UI plus Cocoa programming environment

macOS and iOS Structure

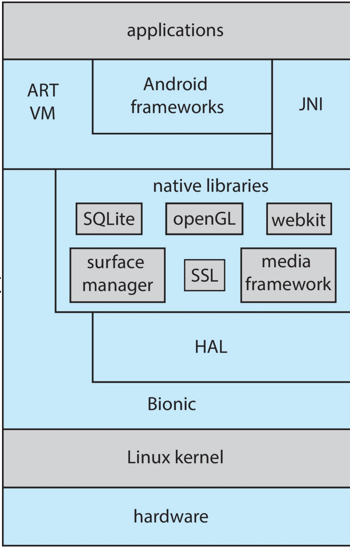
- Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called kernel extensions)





Android

- Developed by Open Handset Alliance (mostly Google) - Open Source
- Similar stack to iOS
- Based on Linux kernel but modified
  - Provides process, memory, device-driver management
  - Adds power management
- Runtime environment includes core set of libraries and Dalvik virtual machine
  - Apps developed in Java plus Android API
  - Java class files compiled to Java bytecode then translated to executable thnn runs in Dalvik VM
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc



In-Class Quiz

Current trends in operating system kernel design favor a hybrid approach. This approach is based on which type of kernel, and what is the main reason driving this trend?

1. Microkernel, driven by the rise of distributed networks.
2. Monolithic kernel, due to security concerns.
3. Layered kernel, due to power efficiency.
4. Modular kernel, due to development complexity.

System Call

Read the Man Pages

- `man 2 syscalls`: lists all system calls. Online: man7: `syscalls(2)`.
- `man 2 syscall`: describes the generic `syscall()` interface. Online: man7: `syscall(2)`.
  - You will see `x86-64 syscall rax ....` This means that on x86/64 the system call number must be placed in `RAX` (64-bit OS) or `EAX` (32-bit OS).
- You can also use Linux syscall table to see that 60 corresponds to `exit`.
- That is why `"mov $60, %eax\n"`

```
void _start() {
    __asm__("mov $60, %eax\n"  // syscall: exit
            "xor %edi, %edi\n" // status: 0
            "syscall");
}
```

The state machine perspective on programs

Program =
Computation $\rightarrow$ `syscall` $\rightarrow$ Computation $\rightarrow$ `syscall` $\rightarrow$ ...

The Right Tool: Trace

What is tracing?

In general, trace refers to the process of following anything from the beginning to the end. For example, the `traceroute` command follows each of the network hops as your computer connects to another computer.

System call trace (`strace`)

- Understand how a program interacts with the OS.
- Observe the system calls made while the program runs.
- Demo: try the smallest possible `Hello World` and inspect its calls.

To observe system calls:

```
$ strace -f gcc helloworld.c
```

- You will see many complex lines. This is normal.
- **Do not give up!**
- Ask ChatGPT for help: “This `strace` output is too complex. Please make it more readable.”
- Refine your question until the explanation is clear.

To observe system calls (Cont.):

Save the output to a text file

```
strace -f gcc helloworld.c 2> gcc_trace.txt
```

Then tell students

- Share `gcc_trace.txt` with ChatGPT and ask for a clear summary.
- This makes the output readable and improves your efficiency.
- **In the AI era, the cost of persistence is small, and the cost of giving up is large.**

Any Program in the Operating System

Any program is a state machine

- The OS always loads the program.
- Another process calls `execve` to set the initial state.
- The program runs as a state machine with computation and system calls.
  - Process management: `fork`, `execve`, `exit`.
  - File and device I/O: `open`, `close`, `read`, `write`.
  - Memory management: `mmap`, `brk`.
- The program finally exits by calling `_exit` or `exit_group`.

Takeaway: browsers, games, antivirus, and malware all use the same OS APIs.

Hands-on: Observe Program Execution

Tool program: the compiler (`gcc`)

- Run `strace -f gcc helloworld.c`. `gcc` starts other processes.
- You can pipe source into an editor like `vim -`. VS Code also works.
- In Vim you can filter text with `:%!grep`.
- The right toolchain matters for developers.

GUI program: the editor (`xedit`)

- Run `strace xedit`.
- A GUI program talks to the X server using the X11 protocol.
- In a VM, `xedit` sends X11 commands through `ssh` with X11 forwarding to the host.

Formula view: Program vs. Operating System

- **Program** = Computation + System Call
- The system call interface is the **bridge** between a program and the operating system.

Operating System = System Call + **?**

Objects

Operating System = System Call + **Objects**

What objects exist in an Operating System?

Objects in the Operating System

Processes

- A process can be viewed as a state machine
- Process management APIs: `fork`, `execve`, `exit`

Contiguous Memory Regions

- We can treat a contiguous memory region as an object
  - Shared across processes
  - Or mapped to files
- Memory management APIs: `mmap`, `munmap`, `mprotect`, `msync`

We will spend about half of the course studying these objects, but the OS certainly has other objects as well!

Files: Named Data Objects

- Byte streams (e.g., terminal, random)
- Byte sequences (regular files)

How to View Devices in Unix?

```
$ ls -l /dev
total 0
lrwxrwxrwx 1 root root   11 Aug 28 14:29 core -> /proc/
   kcore
lrwxrwxrwx 1 root root   13 Aug 28 14:29 fd -> /proc/
   self/fd
crw-rw-rw- 1 root root 1, 7 Aug 28 14:29 full
drwxrwxrwt 2 root root   40 Aug 28 14:29 mqueue
crw-rw-rw- 1 root root 1, 3 Aug 28 14:29 null
lrwxrwxrwx 1 root root    8 Aug 28 14:29 ptmx -> pts/
   ptmx
drwxr-xr-x 2 root root    0 Aug 28 14:29 pts
crw-rw-rw- 1 root root 1, 8 Aug 28 14:29 random
drwxrwxrwt 2 root root   40 Aug 28 14:29 shm
lrwxrwxrwx 1 root root   15 Aug 28 14:29 stderr -> /
   proc/self/fd/2
lrwxrwxrwx 1 root root   15 Aug 28 14:29 stdin -> /proc
   /self/fd/0
lrwxrwxrwx 1 root root   15 Aug 28 14:29 stdout -> /
   proc/self/fd/1
```

Devices in Unix?

The `/dev` directory contains special files that represent devices.

```
crw-rw-rw- 1 root root 1, 9 Aug 28 14:29 urandom
```

The first letter shows the type:

- `c` = character device
- `b` = block device
- `-` = regular file

Let’s take a closer look at these devices

```
$ cat /dev/urandom
$ ls -l /dev/urandom
crw-rw-rw- 1 root root 1, 9 Aug 28 14:29 /dev/urandom
$ touch /tmp/a.c
$ ls -l /tmp/a.c
-rw-r--rw- 1 vscode vscode 0 Aug 28 14:41 /tmp/a.c
```

Question:

Are these (`/dev/null` and `/dev/urandom`) files or devices?

Exercise: File or Device?

They are **both**.

In Unix, Everything is a file.

Devices are also represented as files.

The leading `c` means **character device**: data is consumed or generated as a stream.
For example:

- `/dev/null`: Data written disappears, reading gives nothing.
- `/dev/urandom`: Reading produces an endless stream of random bytes using `cat /dev/urandom`

a.out is also a file

Ask like a ChatGPT: I have an `a.out` file, how can I explore what's inside?

```
$ gcc helloworld.c
$ ./a.out
$ file a.out
$ strings a.out
```

Don't have `file` installed? Run:

```
$ sudo apt-get update
$ sudo apt-get install -y file
```

Computer Science is a human-made science of tools that anyone can copy.

- With AI, your gap to top experts can be very small!

File Descriptors and Pipes

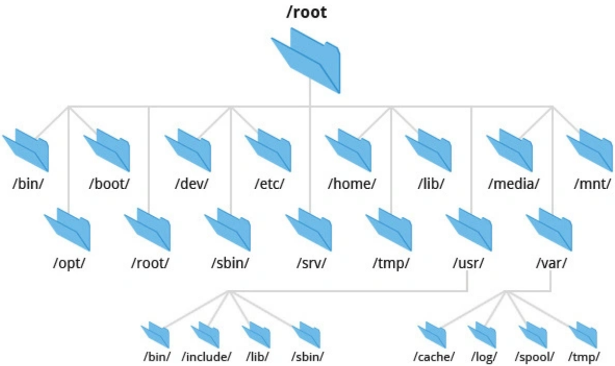
What exactly does `ls /` print?

- `/` root
- `/bin`, `/sbin` — essential user and system programs
- `/usr/bin`, `/usr/sbin` — non-essential programs
- `/lib`, `/lib64`, `/usr/lib` — shared libraries
- `/etc` — system configuration
- `/home` — user home directories
- `/var` — logs, spool, caches
- `/tmp` — temporary files
- `/dev` — device files
- `/proc`, `/sys` — kernel and process views
- `/media`, `/mnt` — mount points
- `/opt` — add-on software

What files does an OS have?

Filesystem Hierarchy Standard (FHS)

- Enables software and users to predict the locations of installed files and directories on Linux systems.
- Defines common directories such as `/bin`, `/etc`, `/usr`, `/var`, `/home`, and others.
- Not every operating system follows FHS. For example, macOS does not conform.



Fun Fact: If the files are right, the OS boots

Idea: Get the right files in the right places and the system will run.

1. Create an EFI System Partition (UEFI) and copy the correct loader.
2. Create a filesystem on the root partition: `mkfs ...`
3. Copy the root filesystem while preserving metadata: `cp -aR SRC/ DST/`
    - Check `/etc/fstab` for correct `UUID`s.
    - You now have a bootable disk.
4. Mount required virtual filesystems at runtime.
    - On disk, `/dev`, `/proc`, `/sys` are empty.
    - Example: `mount -t proc proc /mnt/proc`

That is why a bootable USB works.

Anything Readable or Writable Can Be a File

Real devices

- `/dev/sda`, `/dev/tty`

Virtual devices (special files)

- `/dev/urandom` (random bytes), `/dev/null` (bit bucket)
  - There is no real on-disk file behind them
  - The OS implements custom `read` and `write` handlers
  - Linux source: drivers/char/mem.c
- You can control hardware via `/sys`, e.g., `/sys/class/backlight` to control screen brightness.

Also: `procfs` follows the same idea. Programs use the same APIs to access them.

File Descriptors

- A "pointer" to operating system objects
  - *Everything is a file*
  - Through a descriptor, you can access "everything"
- All object access requires a descriptor
- APIs: `open`, `close`, `read`/`write`, `lseek` (offset manipulation), `dup` (duplicate descriptor)

[`$ man 2 read`]

```
DESCRIPTION           top
       read() attempts to read up to count bytes from
   file descriptor fd
       into the buffer starting at buf.
```

File Descriptors: The "Pointer" to Access Files

- **open**
  - `p = malloc(sizeof(FileDescriptor));`
  - Like `malloc`, **open allocates a new resource** managed by the OS.
- **close**
  - `delete(p);`
  - Like `delete`, **close releases the resource** so it can no longer be used.
- **read/write**
  - `*(p.data++);`
  - Similar to dereferencing a pointer, **read/write moves data through the descriptor**.
- **lseek**
  - `p.data += offset;`
  - Just like changing a pointer offset, **lseek changes the file position**.
- **dup**
  - `q = p;`
  - Like copying a pointer, **dup creates another reference to the same file object**.

Inspecting File Descriptors via `/proc`

Goal: Use the `/proc/<pid>/fd` directory to see what a process is connected to.

```
$ ps
    PID TTY          TIME CMD
   4792 pts/1    00:00:00 bash
   9025 pts/1    00:00:00 cat
  14810 pts/1    00:00:00 ps

$ ls -l /proc/4792/fd
lrwx------ 1 vscode vscode 64 ... 0   -> /dev/pts/1
lrwx------ 1 vscode vscode 64 ... 1   -> /dev/pts/1
lrwx------ 1 vscode vscode 64 ... 2   -> /dev/pts/1
l-wx------ 1 vscode vscode 64 ... 22  -> ~/.vscode-
    remote/.../ptyhost.log
l-wx------ 1 vscode vscode 64 ... 24  -> ~/.vscode-
    remote/.../remoteTelemetry.log
l-wx------ 1 vscode vscode 64 ... 25  -> ~/.vscode-
    remote/.../remoteagent.log
lrwx------ 1 vscode vscode 64 ... 26  -> /dev/pts/ptmx
lrwx------ 1 vscode vscode 64 ... 255 -> /dev/pts/1
```

Inspecting File Descriptors via `/proc`

```
lrwx------ 1 vscode vscode 64 ... 0   -> /dev/pts/1
lrwx------ 1 vscode vscode 64 ... 1   -> /dev/pts/1
lrwx------ 1 vscode vscode 64 ... 2   -> /dev/pts/1
l-wx------ 1 vscode vscode 64 ... 22  -> ~/.vscode-
    remote/.../ptyhost.log
l-wx------ 1 vscode vscode 64 ... 24  -> ~/.vscode-
    remote/.../remoteTelemetry.log
l-wx------ 1 vscode vscode 64 ... 25  -> ~/.vscode-
    remote/.../remoteagent.log
lrwx------ 1 vscode vscode 64 ... 26  -> /dev/pts/ptmx
lrwx------ 1 vscode vscode 64 ... 255 -> /dev/pts/1
```

- Each process exposes its open files under `/proc/<pid>/fd/`.
- `0`, `1`, `2` are **stdin**, **stdout**, **stderr**. Here they point to the terminal `/dev/pts/1`.
- The terminal is a **device file**. Unix treats devices as files.

Inspecting File Descriptors via `/proc`

```
lrwx------ 1 vscode vscode 64 ... 0   -> /dev/pts/1
lrwx------ 1 vscode vscode 64 ... 1   -> /dev/pts/1
lrwx------ 1 vscode vscode 64 ... 2   -> /dev/pts/1
l-wx------ 1 vscode vscode 64 ... 22  -> ˜/.vscode-
    remote/.../ptyhost.log
l-wx------ 1 vscode vscode 64 ... 24  -> ˜/.vscode-
    remote/.../remoteTelemetry.log
l-wx------ 1 vscode vscode 64 ... 25  -> ˜/.vscode-
    remote/.../remoteagent.log
lrwx------ 1 vscode vscode 64 ... 26  -> /dev/pts/ptmx
lrwx------ 1 vscode vscode 64 ... 255 -> /dev/pts/1
```

- Extra descriptors show other connections. Here `22{25` point to log files, `26` to `/dev/pts/ptmx` (the pseudoterminal multiplexer).
- `255` is an internal descriptor created by `bash`. It often duplicates a terminal handle.

How File Descriptors Are Allocated

Rule: The OS assigns the lowest-numbered unused descriptor.

- **0**, **1**, **2** are standard input, output, and error.
- New descriptors usually start at **3**.
- A descriptor is an **index** into the process's file-descriptor table.
- After `close()`, the number can be reused.
- Try It: [4_fd.c]()

How many files can a process open?

- Per-process limit: `ulimit -n` (`ulimit -Hn` for the hard limit).
- System-wide maximum handles: `sysctl fs.file-max` or `cat /proc/sys/fs/file-max`.

Offset in File Descriptors

A file descriptor is part of the process state

- It lives in the kernel. A program accesses it only by an integer index.
- Each file descriptor has its own current offset.

Quiz: After `fork()` and `dup()`, do the file descriptors share the offset?

- Yes. After `fork()` or `dup()`, the file descriptors share the offset.
- But ...
- Try It: 4_fd.c

How the OS avoids overwrite

If the offset were handled poorly

- Two writers could start at the same position.
- New data could overwrite old data.

What the OS actually does

- The kernel keeps one open file description that holds the file offset and flags.
- `dup()` and `fork()` create descriptors that point to the same open file description.
- Each `write()` updates the shared offset atomically, so two writes do not write the same bytes. The order may be different from what you expect.

When you need other behavior

- Open the file again to get an independent offset.
- Use `O_APPEND` to append safely across processes.
- Use `pwrite()` to write at a fixed position without changing the offset.

Windows: File Handles

Handle

- An opaque reference to a kernel object, not a memory pointer.
- Plays a role similar to Unix file descriptors for I/O.
- Created with `CreateFile` and closed with `CloseHandle`.

Windows Process Creation

Engineering-minded design

- **By default handles are not inherited.** In Unix descriptors inherit by default.
  - A child only receives inheritable handles if `CreateProcess(..., bInheritHandles=TRUE, ...)`.
  - Make a handle inheritable at creation: `SECURITY_ATTRIBUTES.bInheritHandle=TRUE`.
  - Or change it later: `SetHandleInformation(h, HANDLE_FLAG_INHERIT, HANDLE_FLAG_INHERIT)`.
- Follow the principle of **least privilege**.

Now, Linux added `close-on-exec` for security.

- Descriptors inherit on `fork`. They stay open on `exec` unless flagged.
- Set at open time: `open(..., O_CLOEXEC)`.
- Or set later: `fcntl(fd, F_SETFD, FD_CLOEXEC)`.

Anything Readable or Writable Can Be a File (cont.)

Pipes: a special kind of "file" (stream)

- Shared by a reader and a writer
- Read end supports `read`; write end supports `write`

Anonymous pipe

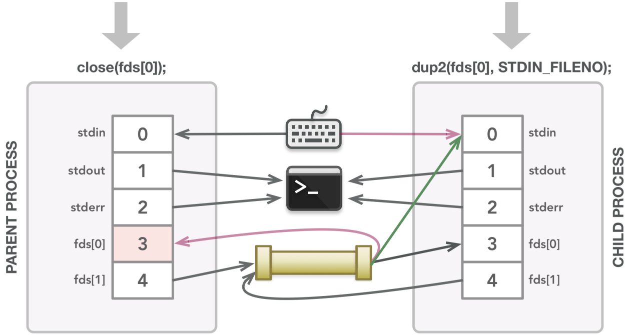
```
int pipe(int pipefd[2]);    // pipefd[0] = read end,
    pipefd[1] = write end
```

- Returns two file descriptors
- One process holds both ends at first
- After `fork()`, parent and child can share them
- Typical use: close unused ends, then connect with `dup2()`

A pipe behaves like a stream. Reading consumes bytes. Writing produces bytes. When all writers close, readers see EOF.

Try It

```
$ ls
$ ls | wc -l
$ touch a.txt
$ ls > a.txt
$ cat a.txt
```



What are file descriptors good for?

Byte streams

- Sequential read and sequential write.
- The reader waits when no data is available.
- Typical example: a pipe.

Byte sequences (random access files)

- Possible but less convenient with plain `read`/`write`.
- You must [`lseek`]() to a position then read or write.
- `mmap` lets you access bytes through a pointer.
- `madvise` and `msync` give finer control.

Rethinking "Everything is a File"

Pros

- Elegant and uniform abstraction.
- Text interfaces are easy to use.

Cons

- Tight coupling with many APIs.
  - Example: a `fork()` in the road.
- Not friendly to high-speed devices.
  - Extra latency and memory copies.
  - Single-threaded I/O path.

Way out: API and Wrapping

Another level of indirection

Any problem in computer science can be solved with another level of indirection.
Butler Lampson

Examples

- Windows NT: Win32 API with a POSIX subsystem. Today we have Windows Subsystem for Linux (WSL).
- macOS: Cocoa API on top of a BSD subsystem.
- Fuchsia: Zircon microkernel with a POSIX compatibility layer.

Limits of compatibility

- No system reaches 100% compatibility.
- Virtual filesystems like `/proc` and `/sys` do not map cleanly.
- WSL1 looked elegant but broke in many real cases.
- Two directions exist: “Windows Subsystem for Linux” and “Linux Subsystem for Windows” (Wine).

Takeaways: Program, OS, and Objects

Program = Computation + System Call
Operating System = Objects + System Call

- The system-call interface links programs and the OS.
- In UNIX, **everything is a file**.
- In UNIX, resources look like files and are controlled by **file descriptors**.
- **Pipes** connect programs by wiring one FD's output to another FD's input.